Controler les transferts de connaissance dans les algorithmes distribués.Application à la détection de l'interblocage

Jean-Michel Hélary, Aomar Maddi, Michel Raynal

INRIA

CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 427

CONTROLER LES TRANSFERTS DE CONNAISSANCE DANS LES ALGORITHMES DISTRIBUÉS

APPLICATION À LA DÉTECTION DE L'INTERBLOCAGE

Jean-Michel HELARY
Aomar MADDI
Michel RAYNAL

Juillet 1985

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Publication Interne n° 260
Juin 1985 - 22 pages

CONTROLER LES TRANSFERTS DE CONNAISSANCE DANS LES ALGORITHMES DISTRIBUES APPLICATION A LA DETECTION DE L'INTERBLOCAGE

Jean-Michel HELARY - Aomar MADDI - Michel RAYNAL

RESUME

Une des caractéristiques fondamentales des algorithmes distribués est l'absence d'une connaissance globale que pourrait capter instantanément un des processus qui réalisent l'algorithme. En conséquence l'élaboration d'un calcul distribué nécessite généralement de mettre à la disposition de chacun des processus des protocoles d'acquisition d'informations (ou connaissances) qu'ont les autres processus. Plusieurs algorithmes distribués ont ainsi été conçus en termes de transferts de connaissances entre processus. Afin de réduire le nombre de messages échangés nous introduisons le concept de connaissance de contrôle et présentons des protocoles fondés sur son utilisation qui réduisent le nombre de transferts.

En guise d'illustration le principe proposé est appliqué à la détection de l'interblocage dans un système distribué. Dans le cas où le graphe des attentes est complet, l'algorithme obtenu ne requiert que 2(n-1) messages pour permettre à l'un des n processus de savoir s'il est ou non définitivement bloqué.

ABSTRACT

Distributed algorithms are fundamentally characterized by the lack of a global knowledge which could be instantaneously cast by a process. Thus, to realize a distributed computation, it is necessary to provide processes with some knowledge acquisition protocols. Several distributed algorithms have been devised in such terms of knowledge transferts. In order to resume the number of exchange messages required by knowledge transferts, we introduce the concept of control knowledge and present protocols based on it which reduce this number.

The concept is then used to devise a distributed deadlock-detection algorithm. When the wait-for graph is complete, the resulting algorithm requires only 2(n - 1) messages to allow one of the processes to know whether it is or not definitively waiting.

CONTROLER LES TRANSFERTS DE CONNAISSANCE DANS LES ALGORITHMES DISTRIBUES.

APPLICATION A LA DETECTION DE L'INTERBLOCAGE

J.M. HELARY, A. MADDI, M. RAYNAL

1. CONTROLER L'ACQUISITION DES CONNAISSANCES

Un algorithme distribué est constitué d'un ensemble fini de processus qui, interconnectés par un certain nombre de voies de communication, s'échangent des informations à l'aide de messages (RAY 85a). Une des caractéristiques fondamentales des algorithmes distribués est l'absence d'un état global que pourrait capter instantanément un processus quelconque (LAM 78). De plus un processus n'a initialement à sa disposition que des informations (ou connaissances) locales et l'élaboration d'un calcul distribué nécessite généralement des protocoles d'acquisition des connaissances qu'ont les autres processus.

1.1. Acquérir des connaissances

Plusieurs algorithmes permettant à un processus d'acquérir de la connaissance qu'il ne possède pas initialement, ont été proposés. On trouve ainsi : dans (BOU 85) un algorithme distribué permettant à chacun des processus, qui ne connait initialement que ses voisins (les processus avec lesquels il peut communiquer par envoi ou réception de messages), d'obtenir la connaissance de tout le graphe des communications ; dans (SCH 85) un algorithme distribué qui, sous les mêmes hypothèses initiales, permet à tout processus d'acquérir les connaissances initiales de chacun des autres ; dans (CHL 85) un algorithme qui permet à tout processus d'acquérir la connaissance d'un état antérieur

de chacun des autres et dont la réunion constitue un état global cohérent (et antérieur) de l'ensemble.

Comme l'illustrent ces exemples, la plupart des problèmes distribués peuvent s'analyser en termes de transferts des connaissances (initiales ou acquises) entre les processus. Les algorithmes qui les résolvent sont développés sur une même technique de base pour réaliser les transferts d'informations et aboutir à l'obtention de la connaissance désirée : il s'agit de la technique dite de la *vague*. (wave algorithms [SCH 85]). Son principe est simple : un processus désireux d'acquérir une certaine connaissance diffuse un message approprié à chacun des processus vers lesquels il peut émettre ; lors de la réception de ce message chacun de ceux-ci le diffuse à chacun de ses interlocuteurs après l'avoir enrichi de la connaissance qu'il possède et ainsi de suite. Les messages circulent ainsi entre les processus transportant les connaissances que ces derniers avaient lorsqu'ils ont été visités, l'acquisition de la connaissance véhiculée par un message étant obtenue par accumulations successives de connaissances locales. Si le graphe dirigé des communications est fortement connexe le processus initiateur du message de demande de connaissances recevra de ses prédécesseurs les informations recherchées.

Le problème essentiel qui se pose dans cette technique est celui de la terminaison. S'il existe des circuits dans le graphe des communications dans lesquels le processus initiateur n'apparait pas, les messages peuvent a priori tourner sur de tels circuits sans s'arrêter. Cela est évité par la constatation suivante : un seul parcours du circuit permet d'accumuler toutes les connaissances qu'ont les processus placés sur ce circuit. En conséquence, lorsqu'un processus reçoit un message de requête, alors qu'il a déjà reçu d'autres messages relatifs à la même requête, il teste si la connaissance véhiculée par ce message est identique à celle qu'il possède déjà (et qu'il a obtenu en ajoutant sa connaissance locale à celle d'un message reçu précédemment), auquel cas aucun des processus placé sur le circuit que vient nécessairement de parcourir le message ne l'a enrichi : il n'est donc pas nécessaire de le diffuser à ses successeurs.

1.2. Contrôler les transferts de connaissance

Dans les algorithmes proposés fondés sur la technique précédente, tout processus transmet à ses successeurs dans le graphe des communications la connaissance qu'il a acquise de ses prédécesseurs, augmentée de la connaissance qu'il possède. L'acquisition de la connaissance recherchée par un processus initiateur donné nécessite un certain nombre de transferts de messages qui peuvent être de taille variable (selon l'enrichissement des connaissances transmises).

Nous proposons une technique qui permet de réduire le nombre de messages échangés en exploitant de manière systématique la connaissance qu'ont les processus de leurs successeurs immédiats. Il est important de remarquer qu'initialement, c'est à dire avant que n'importe quelle communication n'ait eu lieu, chacun des processus n'a à sa disposition que des informations locales : connaissance initiale et successeurs immédiats.

Un processus transmet alors à ses successeurs un message de demande d'informations comportant deux types de connaissance : l'une porte sur la connaissance que cherche à obtenir le processus initiateur (connaissance-données), l'autre porte sur le cheminement de cette connaissance à travers les processus (connaissance de contrôle). Avant de transmettre un message à ses interlocuteurs, un processus enrichit d'une part la connaissance qui intéresse l'initiateur de la demande et d'autre part les informations de contrôle en les complétant par la liste de ses interlocuteurs. A la réception d'un tel message, un processus connait donc un sous-ensemble des processus qui ont reçu ou vont recevoir (cela dépend des vitesses des messages) des informations relatives à la même demande d'informations issue d'un initiateur donné. Selon le genre de connaissance demandée un processus peut alors faire l'économie d'envois de messages à ceux de ses interlocuteurs qui apparaissent déjà dans les processus visités.

Dans ce qui suit nous développons le principe que nous venons d'énoncer en l'appliquant à un problème de contrôle classique : la détection du blocage définitif de tout ou partie d'un système. La deuxième partie présente le problème et la troisième les hypothèses sur le contexte des communications ; la quatrième propose un algorithme fondé sur le principe énoncé, qui permet à un processus de savoir s'il appartient à un ensemble de blocage. La cinquième partie fournit la preuve de l'algorithme.

2. ENSEMBLE DE BLOCAGE

Un système de processus communicants peut être modélisé par un graphe non orienté dont les sommets sont les processus et dont les arcs traduisent la possibilité de communiquer directement qu'ont les couples de processus. Le problème qui nous intéresse est celui du blocage définitif de tout ou partie des processus du système, dû à leur impossibilité de communiquer. Il s'agit donc d'un problème de contrôle d'un calcul distribué. Ce problème de contrôle peut être modélisé par un graphe G = (X,E) orienté dont les processus sont les sommets et tel qu'un arc orienté de P_i vers P_j traduit l'attente par P_i d'un évènement (envoi ou réception de messages par exemple) que seul P_j peut produire. Un tel graphe est appelé graphe des attentes. Si les processus sont dotés de structures de contrôle non-déterministes (HOA 78, ICH 83) il est possible qu'un processus P_i attende l'occurrence d'un évènement parmi plusieurs possibles respectivement productibles par P_j, ..., P_k ; il y aura alors dans le graphe des attentes les arcs (P_i,P_j), ..., (P_i,P_k).

Nous introduisons la notion d'ensemble de blocage pour capter la situation dans laquelle des processus sont définitivement bloqués (interblocage). Rappelons qu'un processus est définitivement bloqué si et seulement si, d'une part il attend un évènement (éventuellement parmi plusieurs possibles) et d'autre part,tous les processus desquels il attend un évènement (de façon non-déterministe) sont également définitivement bloqués. Cette situation est appréhendée par la définition suivante :

> B est un ensemble de blocage du graphe des attentes G=(X,E) si et seulement si :
> (i) $B \subseteq X$
> (ii) $\Gamma(B) \subseteq B$
> (iii) $\forall P_i \in B : \Gamma(P_i) \neq \emptyset$
>
> où Γ désigne la fonction : ensemble des successeurs.

Comme le montre cette définition, la détection d'un interblocage n'est pas équivalente à la détection d'un circuit dans le graphe des attentes (ceci est dû à l'attente non-déterministe d'un évènement parmi plusieurs possibles).

Le problème de la détection des interblocages incluant l'attente non-déterministe a été peu abordé. Dans le cadre d'un système distribué, il se pose de la manière suivante : un processus, dont

l'inactivité est due à l'attente d'évènements productibles par d'autres processus, se pose la question : "suis-je définitivement bloqué ?" ; en d'autres termes, le processus appartient-il à un ensemble de blocage ?

Il est facile de voir qu'un processus P_i appartient à un tel ensemble si et seulement si $\Gamma^*(P_i)$ ne possède pas de points de sortie (sommets sans successeurs). Ici, Γ^* désigne la fermeture reflexo-transitive de Γ c'est à dire P_i lui-même et l'ensemble de tous ses descendants.

En effet, si P_i appartient à un ensemble de blocage B, on a $\Gamma(P_i) \subseteq B$ d'où $\Gamma^*(P_i) \subseteq B$ et par conséquent $\forall P \in \Gamma^*(P_i)$ on a $\Gamma(P) \neq \emptyset$ vu la définition d'un ensemble de blocage.

Réciproquement, si $\Gamma^*(P_i)$ ne possède pas de points de sortie alors :

$\forall P \in \Gamma^*(P_i)$ on a $\Gamma(P) \subseteq \Gamma^*(P_i)$

$\forall P \in \Gamma^*(P_i)$ on a $\Gamma(P) \neq \emptyset$

d'où l'on conclut que $\Gamma^*(P_i)$ est un ensemble de blocage.

Le problème de la détection d'un blocage définitif d'un processus donné se pose donc, du point de vue de ce processus dans un contexte distribué, en termes d'acquisition d'une connaissance qu'il ne possède pas initialement : appartient-il ou non à un ensemble de blocage.

L'algorithme proposé permet de répondre à cette question.

3. HYPOTHESES

Les hypothèses faites ont trait à la communication des messages dans un contexte distribué : il s'agit du sens des communications (hypothèse de connexité) et des propriétés liées aux échanges de messages (hypothèse de comportement).

Le processus demandeur va diffuser sa demande à travers le graphe des attentes. Pour que l'information recherchée lui parvienne il est nécessaire de faire une hypothèse de connexité ; or il n'existe pas a priori de telles hypothèses sur le graphe des attentes qui peut être quelconque. On supposera donc que les arcs du graphe des attentes, bien que orientés, peuvent véhiculer des messages dans les deux sens.

On retrouve cette hypothèse sur les possibilités de transferts dans tous les algorithmes d'acquisition de connaissance lorsque le graphe qui traduit les relations entre processus est quelconque (cf. le calcul diffusant de [DIS 80],[MIC 82],[CHM 82]).

Les délais de transmission des messages entre 2 processus sont supposés quelconques mais finis ; ceci entraine la non-perte des messages. On fait de plus l'hypothèse que les messages ne sont pas altérés.

Les messages reçus par un processus et émis par des processus distincts sont ordonnés arbitrairement s'ils arrivent simultanément ; de plus entre deux processus communicants il n'est fait aucune hypothèse sur le séquencement des messages.

4. L'ALGORITHME DE DETECTION

4.1. Principe : un arbre de contrôle

Un processus qui se pose la question de son appartenance à un ensemble de blocage diffuse à l'aide d'un message de requête une demande de connaissance à ses successeurs dans le graphe des attentes qui l'enrichissent et la transmettent à leurs successeurs respectifs. Au cours du déplacement de cette *vague* un arbre de contrôle est construit, qui permettra de ramener au processus initiateur de la requête la réponse attendue. L'arbre est construit de la manière suivante : lors de la première réception d'une requête un processus mémorise le nom de l'émetteur qui devient son père dans l'arbre et diffuse la requête à ses successeurs qui deviennent ses fils ; un processus qui n'a pas de successeurs répond *non* à son père (il n'est pas bloqué) ; lorsqu'un processus reçoit une requête qu'il a déjà reçue il répond *oui* à l'émetteur ; dès qu'un processus a reçu toutes les réponses de ses fils, il les synthétise et les transmet à son père. Lorsque le père a reçu toutes les réponses attendues il a acquis la connaissance recherchée.

Ce principe de construction d'un arbre de contrôle a été introduit par Dijkstra et Scholten dans [DIS 80] ; il a été abondamment repris et utilisé dans de nombreux algorithmes distribués. Si l'on applique le principe que nous avons énoncé au §1.2. (le contrôle des transferts de connaissance) à la construction de l'arbre de contrôle,

il est possible d'obtenir des algorithmes distribués qui nécessitent moins de messages tout en leur conservant une taille bornée.

Dans [CMH 83] un tel arbre de contrôle est utilisé pour permettre à un processus de savoir s'il est ou non définitivement bloqué. Si le graphe des attentes est complet (tout processus attend un évènement productible par n'importe quel autre processus) le nombre de transferts de messages nécessaires à la détection y est de $n(n-1)$ requêtes et autant de réponse soit un nombre de l'ordre de $O(n^2)$. Dans la même configuration l'algorithme proposé, grâce au contrôle du transfert des connaissances, n'en utilise que $2(n-1)$.

4.2. Messages utilisés

L'algorithme utilise deux types de messages. Ceux du type *requête* véhiculent la demande de connaissance d'un processus donné et la connaissance de contrôle, alors que ceux du type *réponse* transportent les connaissances partielles accumulées. Les premiers transitent dans le sens des arcs du graphe des attentes, les seconds en sens inverse.

La structure d'un message de requête est la suivante : *(requête, z,i)* où la connaissance de contrôle z désigne l'ensemble des processus dont on sait qu'ils ont été ou seront visités par le message de requête, ce qui va permettre de contrôler les transferts. i désigne le processus émetteur du message.

Un message de réponse est structuré ainsi : *(réponse, b)* où b désigne la valeur *oui* ou *non* ; (ces deux valeurs correspondent respectivement aux booléens *vrai* et *faux* et on peut donc leur appliquer les opérateurs booléens pour cumuler la connaissance ainsi transmise).

4.3. Contexte local d'un processus

Chaque processus P_i est doté de variables locales qui définissent son contexte de calcul et de communication.

- $reçu_i$ est un booléen, initialisé à *faux* , qui précise si une requête a déjà été reçue.
- rep_i est une variable à valeurs dans {*oui* , *non*}, initialisée à *oui* ; elle sert à mémoriser la connaissance partielle reçue, relative à la requête.

- $succ_i$ désigne l'ensemble des processus auxquels P_i va transmettre la requête, initialisé à l'ensemble des successeurs de P_i dans le graphe des attentes : il s'agit d'une information locale. C'est avec cette variable que va être mis en oeuvre le principe énoncé : elle sera mise à jour, lors de la réception d'un message *(requête,z,j)* à l'aide de z.
- $pred_i$ sert à mémoriser l'identité du premier processus qui a envoyé un message de requête à P_i ; c'est le champ j de ce message. Cette variable sert à construire l'arbre de contrôle "au vol" et permettra en conséquence d'acheminer la connaissance transportée par les réponses.
- $nbrep_i$ est une variable entière non négative qui compte le nombre de réponses attendues.

Remarque

L'algorithme présenté peut évidemment être adapté au cas où plusieurs processus se posent la question de savoir s'ils appartiennent ou non à un ensemble de blocage ; un même processus peut aussi se la poser plusieurs fois de suite sans attendre d'avoir reçu les réponses correspondantes. Pour cela, toute requête est identifiée de manière unique et les variables locales des processus deviennent des tableaux, chacune des entrées étant relative à un processus initiateur donné. L'unicité de l'identité des requêtes est assurée en considérant le couple : (identité du processus initiateur, numéro de la requête) comme identificateur de requête. L'algorithme sera alors modifié en conséquence (les numéros des requêtes issues d'un processus donné forment une séquence croissante).

4.4. L'algorithme

Lorsque P_i reçoit pour la première fois un message de requête, il réagit de la manière suivante : s'il n'a pas de successeurs, il envoie la réponse *non* à l'émetteur ; s'il en possède il regarde si ses successeurs sont tous dans l'ensemble z , auquel cas il envoie la réponse *oui* à l'émetteur ; s'ils ne sont pas tous dans z il mémorise l'identité de son père dans l'arbre (c'est le processus émetteur) puis diffuse à ceux de ses successeurs qui ne sont pas dans z le message de requête, après avoir mis z à jour ; il attend les réponses de ces processus et dès qu'il les a toutes obtenues (ou dès qu'il peut conclure, voir la remarque après l'algorithme) il renvoie une réponse à son père. Les réceptions ultérieures de messages de requêtes par P_i provoqueront le renvoi immédiat de la réponse *oui* à l'émetteur.

Le texte de P_i est présenté sous forme évènementielle à la manière de [RAY 85b] : l'arrivée de tel message y produit tel effet.

P_{init} désigne le processus initiateur de la demande d'information.

```
lors de reception (requete, z,j)
        faire
1         si reçu alors envoyer (reponse, oui) à P_j
2           sinon
3             reçu := vrai ;
4             si succ = ∅
5                alors (*rep := non*)   envoyer (reponse, non) à P_j
6                sinon
7                    si succ ⊆ z
8                      alors envoyer (reponse, oui) à P_j (*rep=oui*)
9                      sinon
10                      pred := j ;
11                      succ := succ - z ;
12                      nbrep := cardinal (succ) ;
13                      ∀ k ∈ succ : envoyer (requete, z ∪ succ, i) à P_k
                     fsi
                fsi
           fsi
        fait
lors de reception (reponse,b)
        faire
14          rep := rep et b ;
15          nbrep := nbrep - 1 ;
16          si i ≠ init  et nbrep = 0
                 alors envoyer (reponse,rep) à P_pred
17          fsi
18
19      fait
```

P_{init} lance la détection en faisant comme s'il avait reçu le message *(requête, {init} , init)*.

La fin du calcul intervient (aucun message conséquent à la demande de P_{init} ne circule plus) dès que P_{init} a reçu les réponses de tous ses successeurs. L'appartenance de P_{init} à un ensemble de blocage est alors caractérisée par la valeur de la variable rep_{init} à *oui* ou *non*.

Remarque

Cette connaissance peut être acquise par P_{init} avant même la fin du calcul. En effet, si P_{init} reçoit un message *(reponse, non)* d'un de ses successeurs, il peut conclure qu'il n'appartient pas à un ensemble de blocage car la variable rep_{init} prend à ce moment la valeur *non* (ligne 14) et, d'après les propriétés de l'opérateur booleen *et* , ne s'en départira plus : la propriété *rep=non* est stable. Cette remarque est d'ailleurs valable pour n'importe lequel des processus qui attend des réponses ; il est facile de voir que le traitement des messages *reponse* peut être effectué de la manière suivante, équivalente à celle donnée plus haut :

```
lors de reception (reponse , b)
        faire
          si rep = oui
            alors rep := b ;
                  nbrep := nbrep - 1 ;
                  si i ≠ init et nbrep = 0 ou rep = non)
                     alors envoyer (reponse,rep) à P pred
                  fsi
           (*sinon la réponse a déjà été envoyée, lorsque rep est passé à non;
                  ignorer le message*)
          fsi
        fait
```

Il est clair que cette modification apporte une amélioration sur le plan du temps de réponse de l'algorithme (temps d'acquisition de la connaissance par P_{init}), mais ne diminue pas le nombre de messages. Toutefois, les messages de type *reponse* seront purement et simplement ignorés par leur récepteur dès lors que leur variable *rep* aura la valeur *non*.

5. PREUVE

La preuve de l'algorithme consiste à montrer :

- que lorsque le processus initiateur (que nous appellerons P_o dans tout ce qui suit) a lancé une requête pour acquérir la connaissance, il obtient une réponse au bout d'un temps fini (terminaison de l'algorithme)

- et que P_o appartient à un ensemble de blocage si et seulement si la réponse est *oui* (correction partielle).

Afin de démontrer ces deux points, nous introduisons le concept d'arbre d'exécution auquel sont attachées certaines propriétés.

5.1. Arbre d'exécution

Lors d'une exécution de l'algorithme, un arbre de contrôle est construit dynamiquement : il est matérialisé par les variables $pred_i$; selon les vitesses des messages, cet arbre n'est pas nécessairement le même d'une exécution à l'autre. De même, nous considérons un arbre d'exécution A construit de la manière suivante :

i) l'étiquette P_o est associée à la racine
ii) dès qu'un processus P_i reçoit un message *(requête,z,j)*, un noeud d'étiquette P_i est créé dans l'arbre ; il a comme père un noeud d'étiquette P_j (où P_j est le processus émetteur de ce message).

De plus, dans cet arbre d'exécution, un noeud d'étiquette P_i est une feuille si et seulement si un des trois cas suivants a lieu :

i) $reçu_i$ = *vrai* à l'arrivée du message (ligne 1) : un noeud d'étiquette P_i figure déjà dans A.
ii) $reçu_i$ = *faux* à l'arrivée du message (c'est donc le premier message de requête qui parvient à P_i) et $succ_i \subseteq z$ (ligne 7)
iii) $reçu_i$ = *faux* à l'arrivée du message et $succ_i = \emptyset$ (ligne 4)

Dans les deux premiers cas, le processus P_i renvoie immédiatement la réponse *oui* à son père ; nous dirons que P_i est une feuille non-déterminante. Dans le troisième cas, il renvoie immédiatement la réponse *non* à son père : nous dirons que P_i est une feuille déterminante.

Remarquons que l'arbre de contrôle effectivement construit avec les variables $pred_i$ (ligne 10) est un sous-arbre, avec même racine, de l'arbre d'exécution (les feuilles de ce dernier ne sont pas matérialisées dans l'arbre de contrôle).

5.2. Propriétés de l'arbre d'exécution

5.2.1. Proposition 1 Lorsque P_i reçoit un message *(requete,z,j)* la connaissance de contrôle est un ensemble z qui contient :

a) les sommets d'étiquette P_o ,..., P_j , P_i appartenant à un chemin $u = [P_o \ldots P_j\ P_i]$ de G

b) les sommets de $\bigcup \Gamma(P)$

$$P \in [P_o \ldots P_j]$$

Démonstration : elle se fait par récurrence sur la longueur du chemin. Le résultat est vrai pour P_o. Supposons qu'il soit vrai pour P_j, processus qui a envoyé *(requete, z,j)* à P_i.
Alors, d'après les lignes 11 et 13, on a :

i) $P_i \in \Gamma(P_j) - z_j$,(où z_j est la connaissance de contrôle reçue par P_j), sinon P_i n'aurait pas reçu le message.

ii) $z = (\Gamma(P_j) - z_j) \cup z_j$ (connaissance de contrôle envoyée par P_j)

La connaissance de contrôle z reçue par P_i contient donc :

- z_j c'est à dire, d'après l'hypothèse de récurrence :
  - *les sommets P_o ,..., P_l , P_j d'un chemin $[P_o \ldots P_l\ P_j]$
  - *les somments de $\bigcup \Gamma(P)$, $P \in [P_o \ldots P_l]$
- le sommet P_i puisque $P_i \in \Gamma(P_j) - z_j$
- $\Gamma(P_j)$ car $\Gamma(P_j) = (\Gamma(P_j) - z_j) \cup (\Gamma(P_j) \cap z_j) \subset z$

ce qui démontre la proposition 1.

5.2.2. Proposition 2 L'étiquette P_i apparait au moins une fois dans l'arbre d'exécution A si et seulement si $P_i \in \Gamma^*(P_o)$. (Γ^* est la fermeture reflexo-transitive de Γ).

Démonstration : supposons que $P_i \in A$. Si i = 0 alors $P_o \in \Gamma^*(P_o)$. Sinon, soit P_j le père de P_i dans A ; par construction $P_i \in \Gamma(P_j)$; donc, de proche en proche, en remontant la branche de P_i dans A, on obtient $P_i \in \Gamma^*(P_o)$. Réciproquement, montrons par récurrence sur l'entier k que $P_i \in \Gamma^k(P_o) \Rightarrow P_i \in A$. La propriété est vraie pour k = 0

($P_o \in A$). Supposons là vraie jusqu'à k - 1 et soit $P_i \in \Gamma^k(P_o)$; alors $\exists\ P_j \in \Gamma^{k-1}(P_o)$ tel que $P_i \in \Gamma(P_j)$. D'après l'hypothèse de récurrence, $P_j \in A$. Considérons le premier noeud d'étiquette P_j dans A introduit lors de la réception par P_j du premier message *(requete, z, k)*.

On a :
$\Gamma(P_j) = (\Gamma(P_j) - z) \cup (\Gamma(P_j) \cap z)$.
Si $P_i \in \Gamma(P_j) \cap z$ alors P_i est déjà dans l'arbre (d'après la proposition 1) ; si $P_i \in \Gamma(P_j) - z$, il sera introduit dans l'arbre comme fils de P_j (lignes 10, 11, 13) puisque les messages ne se perdent pas. Ceci termine la démonstration par récurrence, et la réciproque est complètement démontrée en observant que, par définition,
$\Gamma^*(P_o) = \bigcup_{k=0}^{\infty} \Gamma^k(P_o)$.

5.2.3. Corollaire 2.1 $\forall P_j \in A$, on a $\Gamma(P_j) \in A$.
Ce résultat a été obtenu au cours de la démonstration de la proposition 2.

5.2.4. Proposition 3 Tout processus $P_i \in \Gamma^*(P_o)$ apparait au plus une fois comme étiquette d'un noeud qui n'est pas une feuille dans A.

Démonstration : Un noeud d'étiquette P_i ne peut engendrer de fils que si, lors de la réception d'un message de requête, le processus P_i vérifie $reçu_i = faux$; or ce booleen est mis à *vrai* à la réception du premier message de requête (ligne 3) et n'est ensuite jamais mis à *faux*.

5.2.5. Corollaire 3.1. lorsqu'un processus $P_i \in \Gamma^*(P_o)$ apparait plusieurs fois comme étiquette dans A, les étiquettes de ses noeuds pères sont toutes différentes.

Démonstration : D'après la proposition 3 un processus ne peut engendrer de fils dans A qu'une seule fois (à la réception du premier message de requêtes). D'après la ligne 13, il envoie alors au plus une requête à chacun de ses successeurs, ce qui démontre le corollaire.

5.3. Terminaison

Théorème 1 : L'algorithme se termine au bout d'un temps fini.

Démonstration : On observe d'abord que le programme de chaque processus (dont le nombre est fini) ne comporte aucune boucle d'itération. Un temps d'exécution non fini ne pourrait donc provenir que de l'échange des messages. Mais d'après la proposition 3 et son corollaire 3.1., l'arbre d'exécution est fini, et par construction, chaque noeud de l'arbre reçoit exactement un message de requête, au bout d'un temps fini puisque les délais de transmission sont finis. Chaque noeud renvoie à son tour une réponse à son père, au bout d'un délai δ. Ce délai est fini. En effet, si le noeud est une feuille, δ est fini ; si le noeud n'est pas une feuille supposons δ infini. Cela signifie que le noeud attend infiniment longtemps une réponse d'un de ses fils, c'est à dire, comme les délais de transmission sont finis, qu'il existe une branche issue de ce noeud sur laquelle chaque processus attend infiniment longtemps. Mais cela est impossible puisque chaque branche est de longueur finie et qu'une feuille répond au bout d'un temps fini.
On déduit de ces observations que P_o reçoit toutes les réponses de ses fils au bout d'un temps fini. CQFD.

5.4. Correction partielle

5.4.1. Proposition 4 Lorsque l'algorithme est terminé, pour tout processus $P_i \in \Gamma^*(P_o)$, les deux propositions suivantes sont équivalentes :

i) $rep_i = non$

ii) un noeud d'étiquette P_i est sur une branche à feuille déterminante.

(Rappelons qu'une feuille est dite déterminante si le processus qui l'étiquette n'a pas de successeurs dans le graphe des attentes (ligne4)). De plus si l'une des 2 propositions est vérifiée, alors P_i n'appartient pas à un ensemble de blocage.

Démonstration :
i) => ii) Si rep_i=*non* il existe au moins une branche passant par un noeud d'étiquette P_i dont la feuille a initialisé la réponse à *non* (ligne 14) ; cette feuille est donc déterminante (la relation rep_i=*non* est stable : une fois vérifiée elle reste toujours vérifiée).

ii) => i) Si un noeud d'étiquette P_i est sur une branche à feuille déterminante ce peut être :

- soit la feuille elle-même, et dans ce cas rep_i est positionné à *non* (ligne 5)
- soit un noeud interne à l'arbre et dans ce cas la feuille déterminante renvoie une réponse *non* ; celle-ci remonte sur la branche jusqu'au noeud d'étiquette P_i et l'on a alors: rep_i = *non* et ... ce qui démontre la proposition.

Si l'une des propositions est vérifiée P_i possède un descendant sans successeur, il n'appartient donc pas à un ensemble de blocage.

5.4.2. proposition 5 (pseudo-réciproque de 4)

Si $P_i \in \Gamma^*(P_o)$ et s'il n'appartient pas à un ensemble de blocage alors l'arbre d'exécution A possède au moins une branche à feuille déterminante. (Elle ne passe pas nécessairement par un noeud d'étiquette P_i).

Démonstration : (par l'absurde)

Si la conclusion est fausse : toutes les branches de l'arbre sont à feuilles non-déterminantes. Dans ce cas l'ensemble B des processus apparaissant dans l'arbre vérifie :

$\forall P \in B \quad \Gamma(P) \neq \emptyset$ (il n'y a pas de feuille déterminante)

$\forall P \in B \quad \Gamma(P) \subseteq B$ (corollaire 2.1)

B est donc un ensemble de blocage ; de plus P_i apparait dans l'arbre d'après la proposition 2 et $P_i \in \Gamma^*(P_o)$ ce qui fournit la contradiction avec les hypothèses. CQFD.

5.4.3. Théorème 2 (correction partielle)

Lorsque l'algorithme se termine on a pour le processus P_o initiateur de la demande de connaissance :

rep_o=*non* <=> $P_o \notin$ à un ensemble de blocage.

Démonstration : si rep_o=*non*, d'après la proposition 4 P_o n'appartient pas à un ensemble de blocage.

Reciproquement si P_o n'appartient pas à un tel ensemble l'arbre d'exécution possède au moins une branche à feuille déterminante (proposition 5). La feuille de cette branche initialise une réponse à *non* qui remonte le long de la branche jusqu'à P_o (qui est racine de l'arbre) et donc, à la terminaison : rep_o = *non*. CQFD.

6. CONCLUSION

Nous avons proposé une technique générale d'acquisition de connaissances basée sur le contrôle des transferts ; elle permet de réduire le nombre de messages échangés par les différents processus qui réalisent un algorithme distribué. Il est important de remarquer que la taille des messages reste bornée.

Le principe sur lequel repose la technique proposée est très simple : si ce n'est pas nécessaire, ne pas refaire une chose qui a déjà été faite. (On retrouve un principe analogue en algorithmique séquentielle : programmation dynamique,...). Il est ici appliqué aux transferts de messages.

Ce principe pourra être appliqué dans d'autres situations, notamment dans les problèmes d'acquisition de connaissance par un processus, à condition que les opérateurs appliqués au type de connaissance soient associatifs, commutatifs (l'ordre dans lequel les connaissances partielles sont accumulées et renvoyées est inopérant sur la connaissance finale) et idempotents (agréger à une connaissance partielle une connaissance qu'elle contient déjà ne modifie pas cette connaissance partielle).

L'algorithme de détecteur d'interblocage proposé illustre l'utilisation de ce principe. Les messages y sont au maximum de taille n (nombre de processus) et leur nombre est toujours inférieur -ou au pire égal- à ceux qu'offrent les autres algorithmes de détection. De plus, grâce au type de la connaissance recherchée par un processus (booleen) la réponse lui parvient "au plus tôt" (en d'autres termes elle peut lui parvenir avant que la détection ne soit entièrement terminée). Cela est possible lorsque, outre les propriétés d'associativité et commutativité évoquées plus haut, le type de la connaissance recherchée possède un élément absorbant (ici la réponse *non*).

REFERENCES-

[BOU 85] BOUGE L.
Symmetry and Genericity for CSP Distributed Systems.
Rapport de Recherche, LITP, Université de Paris 7, (Mai 1985), 22p.

[CHL 85] CHANDY K.M., LAMPORT L.
Distributed Snapshots : determining global states of distributed systems.
ACM TOCS, Vol. 3, 1, (February 1985), pp 63-75.

[CHM 82] CHANDY K.M., MISRA J.
Distributed Computation on graphs : Shortest Paths Algorithms.
Comm. ACM, Vol. 25,11 (November 1982), pp. 833-837.

[CMH 83] CHANDY K.M., MISRA J., HAAS L.M.
Distributed Deadlock Detection.
ACM TOCS, Vol. 1,2 (April 1983), pp. 144-156.

[DIS 80] DIJKSTRA E.W., SCHOLTEN C.S.
Termination Detection for Diffusing Computations.
Inf. Proc. Letters, Vol. 11, 1, (August 1980), pp 1-4.

[HOA 78] HOARE C.A.R.
Communicating Sequential Processes.
Comm. ACM, Vol. 21,8, (August 1978), pp. 666-677.

[ICH 83] ICHBIAH J. et al.
Reference Manual for the ADA Programming Language.
ANSI/MIL-STD, 1815A, (January 1983).

[LAM 78] LAMPORT L.
Time, Clocks and the Ordering of Events in a Distributed System.
Comm. ACM, Vol. 21, 7, (July 1978), pp. 558-565.

[MIC 82] MISRA J., CHANDY K.M.
Termination Detection of Diffusing Computations in CSP.
ACM TOPLAS, Vol. 4,1, (January 1982), 37-43.

[RAY 85a] RAYNAL M.
Towards a better understanding of Distributed Algorithms.
Proc. of an Int. Workshop on Distributed Computing,
Newcastle, (April 1985).

[RAY 85b] RAYNAL M.
Algorithmes Distribués et Protocoles.
A paraître, Eyrolles, (Octobre 1985).

[SCH 85] SCHNEIDER F.B.
Paradigms for distributed programs.
in Distributed Systems, Springer-Verlag, LNCS 190, (1985),
pp 431-480.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique